Theoretical Computer Science 412 (2011) 5527–5540

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2

Volker Turau *, Bernd Hauck

Hamburg University of Technology, Institute of Telematics, Schwarzenbergstrasse 95, D-21073 Hamburg, Germany

ARTICLE INFO

Keywords:
Self-stabilizing algorithms
Approximation algorithms
Weighted matching
Distributed algorithms

ABSTRACT

The maximum weight matching problem is a fundamental problem in graph theory with a variety of important applications. Recently Manne and Mjelde presented the first self-stabilizing algorithm computing a 2-approximation of the optimal solution. They established that their algorithm stabilizes after $O(2^n)$ (resp. $O(3^n)$) moves under a central (resp. distributed) scheduler. This paper contributes a new analysis, improving these bounds considerably. In particular it is shown that the algorithm stabilizes after $O(nm)$ moves under the central scheduler and that a modified version of the algorithm also stabilizes after $O(nm)$ moves under the distributed scheduler. The paper presents a new proof technique based on graph reduction for analyzing the complexity of self-stabilizing algorithms.

1. Introduction

A self-stabilizing distributed system has the ability to tolerate arbitrary transient faults. An algorithm is self-stabilizing if it can start in any possible configuration, gain consistency in a finite number of steps by itself without any external intervention, and remains in a consistent state [3]. The state of a consistent or fault free self-stabilizing system is defined by a predicate based on the state of the system, i.e. on the states of all nodes.

Self-stabilizing algorithms have been applied to different fields such as device drivers, operating systems, and wireless sensor networks [4,27,25]. The majority of research has focused on distributed algorithms for optimization problems in graph theory such as coloring, the minimal dominating set, and the maximal independent set [7,24]. Currently there are few self-stabilizing algorithms for optimization problems with guaranteed approximation ratio, e.g. [13,14]. Recently, Manne and Mjelde [17] presented the first self-stabilizing algorithm for computing a *maximum weighted matching* of a graph with an approximation ratio of 2, i.e., the weight of the computed matching is at least half the weight of the maximum weight. The algorithm uses the shared memory model with composite atomicity [3]. For a central scheduler a bound of $O(2^n)$ moves and for the distributed scheduler a bound of $O(3^n)$ moves have been proven.

The precise determination of the stabilization time of this algorithm remained an open problem. No example for which the algorithm requires an exponential number of moves was given. In this paper we contribute a new analysis of Manne and Mjelde's algorithm and limit the number of moves to $O(nm)$ for the central scheduler. Furthermore, a modified version of this algorithm is presented, requiring $O(nm)$ moves for the unfair distributed scheduler.

An important contribution of this paper is a new technique for computing the move complexity of self-stabilizing graph optimization algorithms. The main idea is to map an execution sequence for a graph to that of a given subgraph. This mapping allows us to derive an upper limit for the difference of the numbers of moves for both execution sequences — for the original graph and for the subgraph. Hence, the total number of moves required for the original graph is bounded by the sum of this

* Corresponding author. Tel.: +49 40 42878 3530.
E-mail addresses: turau@tuhh.de (V. Turau), hauck@tuhh.de (B. Hauck).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.11.032

limit and the number of moves required for the subgraph. Since the subgraph has fewer edges or nodes, the latter number can be determined by induction.

The rest of this paper is organized as follows. Section 2 reviews the state of the art of distributed algorithms for matching problems. Section 3 introduces the definitions and notation used in this paper. Section 4 presents the basic algorithm. Section 5 deals with the synchronous scheduler. The other schedulers are treated in Sections 6 and 7. The paper ends with two conjectures.

2. Related work

Algorithms solving the maximum matching problem received a lot of attention since the early work of Edmonds [5]. This research has been carried out for bipartite and general graphs both in the weighted and unweighted setting. While there are may sequential algorithms, only a small number of distributed algorithms for matching have been proposed [26]. In fact, we are not aware of any distributed algorithm that solves the maximum matching problem optimally, except for special graph classes such as bipartite graphs. Therefore, research has concentrated on finding maximal matchings and on approximating maximum matchings, and the weighted and the unweighted cases. In the following, related work is classified into studies of synchronous and asynchronous systems.

First, distributed algorithms for approximating maximum weighted matchings in synchronous systems are considered. Wattenhofer et al. present a randomized 5-approximation algorithm taking $O(\log n)$ rounds [26]. Nieberg's algorithm computes a $(1+\epsilon)$-approximation in $O(\log n)$ rounds [21]. The unweighted maximum matching problem received considerably more attention. The currently best algorithms for finding approximately optimal matchings are due to Lotker et al. [16]. For any $\epsilon > 0$ they give a randomized distributed $(4+\epsilon)$-approximation algorithm for maximum weighted matching, whose running time is $O(\log n)$ and for unweighted dynamic graphs, they give a distributed algorithm that maintains a $(1+\epsilon)$-approximation in $O(1/\epsilon)$ time for each node insertion or deletion. Since the focus of this paper is on asynchronous systems, readers are referred to the survey of Elkin for a review of the state of the art for synchronous systems [6].

As regards asynchronous systems we first treat the unweighted case. Early work concentrated on the maximal matching problem. The history of self-stabilizing algorithms for the unweighted maximal matching problem goes back to Hsu and Huang [12]. Their algorithm assumes the shared memory model with composite atomicity and a central scheduler and requires $O(n^3)$ moves. Later the analysis of the algorithm was improved and lower bounds were proven of $O(n^2)$ [23], $O(m)$ [10]. Note that the algorithm does not work with a distributed scheduler. Chattopadhyay et al. developed an algorithm that stabilizes in $O(n^2)$ steps for a fair distributed scheduler using the shared memory model with read/write atomicity [2]. Later Manne et al. [19] presented an algorithm that stabilizes in $O(m)$ steps using an unfair distributed scheduler and the shared memory model with composite atomicity.

Whereas the algorithm of Hsu and Huang assumed an anonymous network, these two algorithms require node identifiers that are unique within distance 2. A deterministic self-stabilizing algorithm for the maximal matching problem in anonymous networks is impossible under a synchronous scheduler. For example, let G be a cyclic graph where all edges are of the same weight and the nodes do not have identifiers. If all nodes are in the same state initially, this property will hold after every step. Thus, when an algorithm stabilizes, no node and no edge stands out. Chattopadhyay et al. presented a randomized algorithm for the maximal matching problem in an anonymous network with read/write atomicity [2]. Several methods for transforming algorithms using strong model assumptions to algorithms using weaker assumptions are described in the literature: from a fair scheduler to an unfair scheduler [15], from a central to a distributed scheduler [8] and for atomicity refinement [20,1]. In general, algorithms developed for a specific model are superior to transformed algorithms in terms of complexity.

Approximation of a maximum matching for the unweighted case also received some attention. Manne et al. presented the first self-stabilizing algorithm for finding a 3/2-approximation to this problem using at most exponential time under a distributed adversarial scheduler [18]. This work is done for the shared memory model with composite atomicity.

Finally, the case of maximum weight matchings in asynchronous systems is considered. The work in this area is sparse. Manne and Mjelde developed the first self-stabilizing 2-approximation algorithm for the maximum weight matching problem [17]. The authors showed that their algorithm stabilizes after $O(2^n)$ (resp. $O(3^n)$) moves under a central (resp. distributed) scheduler. They assume unique identifiers and the shared memory model with composite atomicity.

The survey of Guellati and Kheddoucib contains more references for self-stabilizing algorithms solving the matching problem [9].

3. The model

The objective of a self-stabilizing distributed algorithm is to recover from transient faults in bounded time without external intervention. The absence of faults is defined by a predicate $\mathcal{P}$ over the global state of the system. A distributed system consists of a set of processes where two adjacent processes can communicate with each other. As in [17] we assume a shared memory model with *composite atomicity*. Thus, reading the neighbors' states and updating its own state are considered as an atomic action. In contrast to this the read–write atomicity model treats a single read and a single write operation as atomic actions. The latter model is the more general one, but there exist methods for transforming algorithms from one model to the other [3].

The communication relation is represented by an undirected graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, where each process is represented by a node in V, and two processes v_i and v_j are adjacent if and only if $\langle v_i, v_j \rangle \in E$. The set of neighbors of a node $v \in V$ is denoted by $N(v)$. A node v maintains a set of variables, their values represent the *state* s_v of the process. Each variable ranges over a fixed domain of values. A *configuration* c of the graph G is defined as the n-tuple of all nodes' states: $c = (s_{v_1}, \ldots, s_{v_n})$. A configuration $c \in \Sigma$ is called *legitimate* relative to $\mathcal{P}$ if c satisfies $\mathcal{P}$. Hence, a legitimate configuration is free of faults. The set of all configurations in G will be referred to as C^G.

A self-stabilizing algorithm is specified by a set of *rules* of the form

$$\mathit{Name} :: [\mathit{precondition}] \longrightarrow \mathit{statement}$$

The precondition of a rule is a Boolean predicate defined on the state of the node itself and its neighbors. The statement updates the state of the node only. The execution of the statement of a rule is called a *move*. A rule or a move is called *enabled* in a configuration c if its precondition evaluates to true in c. A node is enabled if at least one of its rules is enabled.

Self-stabilizing algorithms operate in *steps*. At the beginning of every step, all nodes check the preconditions of their rules. Then a *scheduler* selects a subset of the enabled nodes to make a move. Common schedulers are the central scheduler (only a single node makes its move in every step), the distributed scheduler (any nonempty subset of the enabled nodes can make their moves simultaneously), and the synchronous scheduler (all enabled nodes make their moves simultaneously). Although it is easier to prove stabilization for algorithms working under the central scheduler, the synchronous and the distributed scheduler are more suitable for practical implementations. Note that the distributed scheduler subsumes schedulers of the other two types and is the most general concept.

In general, schedulers have no restrictions on their scheduling policy. They may choose not to schedule a continuously enabled node to make a move. Therefore they are called *unfair* schedulers. A scheduler is called *fair* if the scheduler prevents a node being continuously enabled without making a move. The results presented in this work are valid for the unfair scheduler.

More formally, a move is a tuple $(s, s')_{v_i}$, where s (resp. s') denotes the state of v_i before (resp. after) the move. If the executing node is clear or of no relevance, the subscript will be omitted. Denote by $\mathcal{M}^G$ the set of all moves of G.

Likewise, a step is a tuple (c_0, c_1), where c_0, c_1 are configurations such that

- all nodes that make a move in this step are enabled in configuration c_0, and
- c_1 is the configuration reached after these nodes have executed a rule in configuration c_0.

When the central scheduler is used, each step consists of a move of a single node only. Thus, if a step consists of the move $\mathfrak{m} = (s, s')$ that transforms configuration c_0 into c_1 we also write $\mathfrak{m} = (c_0, c_1)$ and with a slight abuse of notation $\mathfrak{m}(c_0) = c_1$. This notation does not introduce any ambiguity when the central scheduler is used, since c_0 and c_1 coincide in all components but one.

An *execution* is a maximal sequence $c_0, c_1, \ldots$ of configurations such that for each configuration c_i the next configuration c_{i+1} is obtained from c_i by a step. Let $\mathcal{L}_{\mathcal{P}} \subseteq C^G$ be the set of all legitimate configurations with respect to a predicate $\mathcal{P}$. An algorithm is *self-stabilizing* with respect to $\mathcal{P}$ if the following two conditions hold:

(a) Closure property: If (c_0, c_1) is a step with $c_0 \in \mathcal{L}_{\mathcal{P}}$ then $c_1 \in \mathcal{L}_{\mathcal{P}}$.
(b) Convergence property: For every execution $c_0, c_1, \ldots$ there is an integer i such that $c_i \in \mathcal{L}_{\mathcal{P}}$.

A standard measure for evaluating the complexity of self-stabilizing algorithms is the move complexity. It determines the maximum number of individual moves needed to reach a legitimate configuration. This is relevant for many practical applications such as wireless systems with bounded resources. The execution of self-stabilizing algorithms defined for the composite atomicity model in a wireless setting requires a transformation. The cached Sensornet transform (CST) proposed by Herman is a widely used transformation technique [11]. It requires that nodes broadcast their state to their neighbors after every move. Since communication is the main consumer of energy, a reduction of the number of broadcasts prolongs the lifetime of a network.

In the following the symbol $\perp$ denotes the *empty move*; this move does not change the state of any node. Every node is at any time enabled with respect to the empty move. It is not part of the algorithm under consideration; a convenience for the proof.

4. Maximum weight matching

Let $G = (V, E)$ be a undirected graph, with $n = |V|$ and $m = |E|$. A set M of independent edges of G is called a *matching* of G. M is a *maximal matching* if there is no matching M' with $M \subset M'$. A matching M is a *maximum matching* if there does not exist any matching with cardinality larger than $|M|$. Let G be a weighted undirected graph, the *weight* of an edge e is denoted by $w(e) \in \mathbb{R}_+$. The weight of a matching M is the sum of the weights of all edges of M. A matching is called a *maximum weight matching* if its weight is the maximum among all matchings of G.

Algorithm 1 is a sequential greedy algorithm that calculates a matching M_{greedy} with at least half the weight of the maximum weight [22]. The idea is to start with an empty set and then add the remaining heaviest edge each time. M_{greedy} is unique if the edges' weights are pairwise different. The algorithm of Manne and Mjelde computes M_{greedy} [17].

Algorithm 1 Greedy 2-Approximation of Maximum Weight Matching

$M = \emptyset$
for $\langle u, v\rangle$ **in** E *in descending order with respect to their weight*
  if *neither* u *nor* v *are incident to an edge in* M
    then $M := M \cup \{\langle u, v\rangle\}$

Having unique node identifiers permits the assumption that all edge weights are different. This can be achieved by the following simple definition of a total order on the set of edges. Let $\langle u_1, u_2\rangle$ and $\langle v_1, v_2\rangle$ be two edges; then $\langle u_1, u_2\rangle < \langle v_1, v_2\rangle$ if and only if

- $w(\langle u_1, u_2\rangle) < w(\langle v_1, v_2\rangle)$ or
- $w(\langle u_1, u_2\rangle) = w(\langle v_1, v_2\rangle) \wedge \min(u_1, u_2) < \min(v_1, v_2)$ or
- $w(\langle u_1, u_2\rangle) = w(\langle v_1, v_2\rangle) \wedge \min(u_1, u_2) = \min(v_1, v_2) \wedge$ $\max(u_1, u_2) < \max(v_1, v_2)$.

For the rest of this paper it is assumed that the weights of all edges are pairwise different.

In the following the self-stabilizing algorithm of Manne and Mjelde is presented with a slightly different notation. The state $s_v = (v.p, v.w)$ of a node v is defined by two variables p and w. The intention of these variables is as follows: p stores a pointer (i.e. the identifier) to a neighbor of v or *null*, and w stores the weight of the edge $\langle v, v.p\rangle$, i.e., $w(\langle v, v.p\rangle)$. The definition of $w(\langle\cdot,\cdot\rangle)$ is extended such that $w(\langle v, null\rangle) = 0$. To express that $v.p = u$, we say *node* v *points to node* u or synonymously *node* v *points to edge* $\langle v, u\rangle$.

Let $C_v = \{v_i \in N(v) \mid w(\langle v_i, v\rangle) \geq v_i.w \vee v_i.p \in \{v, null\}\}$. A node $v_{max} \in C_v$ is called *maximal* if $w(\langle v_{max}, v\rangle) \geq w(\langle v_i, v\rangle)\ \forall v_i \in C_v$. If $C_v \neq \emptyset$ then denote by $maxC_v$ the unique maximal node of C_v. The complete algorithm from [17] is depicted below. Note that the definition of C_v has been slightly altered compared to that for the original paper. It is straightforward to see that the results of [17] still hold and the calculated matching is the same.

Algorithm 2 Self-Stabilizing 2-Approximation of Maximum Weight Matching

Functions:
BestMatch(v) :
  if $C_v \neq \emptyset$ **then return** $maxC_v$
  else return *null*

Actions:
$R_1 :: [v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle)] \longrightarrow$
  $v.p := BestMatch(v), v.w := w(\langle v, v.p\rangle)$

Two nodes are called *matched* if they both point at each other. An edge $\langle v, w\rangle$ is matched if v and w are matched. Denote by M the set of all matched edges of a configuration. A node v is called in *sync* if $v.w = w(\langle v, v.p\rangle)$. In [17] it is proved that Algorithm 2 stabilizes under a distributed scheduler with at most $O(3^n)$ moves and that in a configuration of G where no node is enabled, M is a 2-approximation of the maximum weight matching of G. The following sections prove that the move complexity of Algorithm 2 is in fact polynomial.

Definition 1. A configuration c satisfies $\mathcal{P}$ if all nodes are in sync, all nodes not contributing to the matching point to *null* and the matching defined by c is M_{greedy}.

Lemma 1. *There is a unique configuration in which all nodes are disabled with respect to Algorithm 2 and this configuration satisfies* $\mathcal{P}$.

Proof. Consider a configuration c where all nodes are disabled with respect to Algorithm 2. Since rule R_1 is not enabled, all nodes are in sync. Consider a node v with $v.p = u$ but $u.p \neq v$. If $u.p = null$ or $w(\langle u, u.p\rangle) < w(\langle v, u\rangle)$ then u is enabled to point to v. Hence, $w(\langle u, u.p\rangle) > w(\langle v, u\rangle)$ and, thus, v is enabled. This contradiction shows that all nodes not contributing to the matching point to *null* and c defines a matching M_c.

Assume $M_c \neq M_{greedy}$. Let $e_1 = \langle u_1, v_1\rangle$ be the heaviest edge with $e_1 \in M_{greedy}$ and $e_1 \notin M_c$. Then, in configuration c nodes u_1 and v_1 do not point towards each other but also they do not point to heavier edges, since there are no heavier edges leading towards nodes that are not matched via even heavier edges already. Therefore u_1 and v_1 are enabled, contradicting the assumption. □

A disadvantage of the algorithms presented must be mentioned. A configuration that represents a maximum weight matching is not necessarily a legitimate configuration, i.e. it does not necessarily satisfy $\mathcal{P}$. To see this, consider a graph consisting of the nodes v_1, v_2, v_3, and v_4 and the three edges $\langle v_1, v_2\rangle$, $\langle v_2, v_3\rangle$ and $\langle v_3, v_4\rangle$ with weights 1, $1 + \epsilon$, and 1 where $\epsilon > 0$. A maximum weight matching consists of the edges $\langle v_1, v_2\rangle$ and $\langle v_3, v_4\rangle$ with a total weight of 2. Even if initialized with this matching, Algorithm 2 will stabilize with the non-maximum weight matching consisting of the edge $\langle v_2, v_3\rangle$ with a total weight of $1 + \epsilon$.

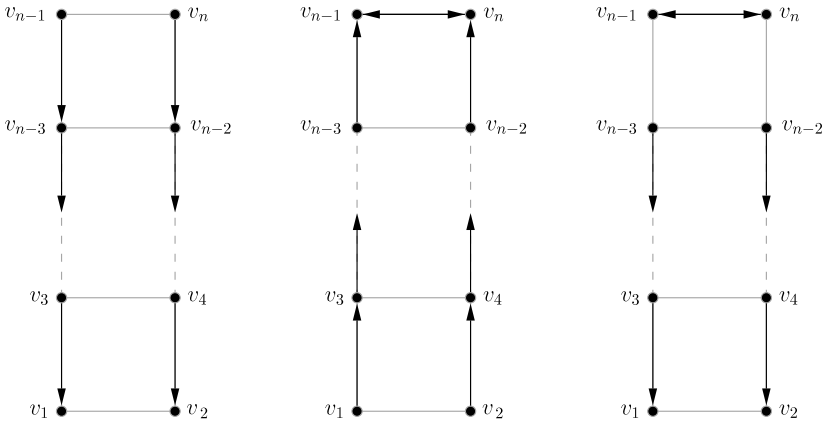


Fig. 1. Start of the execution of Algorithm 2 under the synchronous scheduler for graph L.

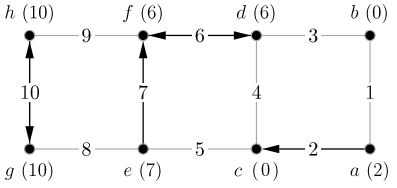


Fig. 2. Φ-edges $\langle a, c\rangle$, $\langle e, f\rangle$ and $\langle g, h\rangle$.

5. A synchronous scheduler

An upper bound of $O(n^2)$ moves until stabilization occurs for Algorithm 2 executed under a synchronous scheduler directly follows from [17]. This section provides an example that shows that this is also a lower bound. For this purpose consider the *ladder graph* L with n nodes; let n be even and $\langle v_i, v_j\rangle \in E$ if $j = i + 2$, or $j = i + 1$ and i is odd. The weights of the edges satisfy the relation $w(\langle v_{i_1}, v_{j_1}\rangle) < w(\langle v_{i_2}, v_{j_2}\rangle)$ if

- $min(i_1, j_1) < min(i_2, j_2)$, or
- $min(i_1, j_1) = min(i_2, j_2) \wedge max(i_1, j_1) < max(i_2, j_2)$.

Fig. 1 gives an initial configuration and shows the first two steps of an execution of Algorithm 2. The nodes' pointers are indicated by arrows. In the final configuration the matching consists of all edges $\langle v_i, v_j\rangle$, where i is odd and $j = i + 1$. This configuration is reached after $n^2/2$ moves.

6. A central scheduler

6.1. Φ-edges

We define the function $\Phi : C^G \longrightarrow \mathbb{R}_+$ as the sum of the weights of all edges $\langle x, x.p\rangle$ that meet the following conditions:

- x is disabled with respect to Algorithm 2, and
- if $x.p$ is enabled, then its move will not enable x.

In other words: An edge contributes to Φ iff either both nodes are disabled and pointing to each other or they are exactly one move away from becoming so. An edge $\langle x, x.p\rangle$ contributes to Φ only once, even if $x.p$ points to x itself and the rules above also apply to the edge $\langle x.p, x\rangle$. Edges that contribute to the value of function Φ will be called Φ-*edges*. This status of an edge $\langle v, w\rangle$ with respect to being a Φ-edge can only change after a move of a neighbor of v or w.

Fig. 2 illustrates the concept of Φ-edges. The nodes' pointers are indicated by arrows; the values of their weight variables are in brackets. Edge $\langle a, c\rangle$ is a Φ-edge, because a points to c and is disabled while c is enabled to point to a; g and h are matched and disabled and, thus, $\langle g, h\rangle$ also contributes to Φ. Nodes d and e are both disabled and point towards f. However, f is enabled to point to e, so $\langle d, f\rangle$ is not a Φ-edge, but $\langle e, f\rangle$ is.

Lemma 2. *Φ is monotonically increasing and so is the number of Φ-edges.*

Proof. The value of Φ may change due to the nodes' moves. We analyze the impact of a nodes' moves on Φ one by one. Let x be any node that is enabled with respect to Algorithm 2. On executing its move, node x changes its pointer from $x.p_{old}$ to $x.p_{new}$.

If $x.p_{old} = x.p_{new}$ only the weight variable of x changed but its pointer did not. Any neighbor $y \neq x.p_{new}$ of x with $y.p = x$ gets enabled and, thus, $\langle y, x\rangle$ did not contribute to Φ before. If $x.p_{new}$ is enabled to point to x after that move, $\langle x, x.p_{new}\rangle$ is a

new Φ-edge. So let $x.p_{old} \neq x.p_{new}$. We distinguish two cases that are to be applied after the move of x:

Case 1: $x.p_{new} = null$.

Φ remains unchanged, since x was enabled before and all its neighbors are pointing to edges heavier than $\langle x, x.p_{old}\rangle$; otherwise x would not have been enabled to perform this move. None of the neighbors gets enabled or disabled by x's move.

Case 2a: $x.p_{new} \neq null$ and $x.p_{new}$ is disabled.

Clearly $x.p_{new}$ was pointing to x and $\langle x.p_{new}, x\rangle$ was a Φ-edge before x's move. Hence, even though $\langle x, x.p_{new}\rangle$ meets all conditions for being a Φ-edge it does not increase the Φ-value. Nodes that are pointing towards x have a lower weight than $w(\langle x.p_{new}, x\rangle)$. So they did not contribute to Φ anyway. However, Φ can increase if, due to the move of x, a neighbor $y \in N(x)$ that also pointed to x before (e.g. $y = x.p_{old}$ or $x.p_{old} = null$) becomes enabled to point to a disabled node z that on its part already points to y.

Case 2b: $x.p_{new} \neq null$, $x.p_{new}$ is enabled, and its move would not enable x.

Note that $\langle x, x.p_{new}\rangle$ is a Φ-edge. We have to show that the weight of this edge is heavier than the weight of an edge that previously was a Φ-edge but lost this status due to the move of x. We distinguish two cases:

- $w(\langle x, x.p_{new}\rangle) > w(\langle x, x.p_{old}\rangle)$.
  If there have been other nodes pointing at x (resp. $x.p_{new}$) before x's move, their edges would have been of lower weight than $\langle x, x.p_{new}\rangle$ and they would not have been taken into account for Φ, since the move of x (resp. $x.p_{new}$) enables them. Other nodes do not become enabled. Thus, no Φ-edge loses this status. Since $\langle x, x.p_{new}\rangle$ is a new Φ-edge, the value of Φ increases.
- $w(\langle x, x.p_{new}\rangle) < w(\langle x, x.p_{old}\rangle)$.
  $x.p_{old}$ was pointing to another node with higher weight and so $w(\langle x, x.p_{old}\rangle)$ did not account for Φ. Now x is the node that points to $x.p_{new}$ with heaviest weight. If there have been other nodes pointing at x before x's move, their edges would have been of lower weight than $\langle x, x.p_{new}\rangle$ and they would not have been taken into account for Φ, since the move of x enables them. If there are other nodes pointing at $x.p_{new}$, one of its edges could have been a Φ-edge before the move. Due to the move of x this edge loses this status, since $x.p_{new}$ is enabled to point to x. However, the value of Φ increases, because the new Φ-edge, $\langle x, x.p_{new}\rangle$, has a higher weight than the former Φ-edge.

Case 2c: $x.p_{new} \neq null$, $x.p_{new}$ is enabled, and its move would enable x.

Note that $\langle x, x.p_{new}\rangle$ is not a Φ-edge. $x.p_{new}$ was enabled to point to an edge of weight greater than $\langle x, x.p_{new}\rangle$ before the move by x. So $x.p_{new}$ is not affected by the move of x and nor are the neighbors of $x.p_{new}$. x obviously was enabled before its move and so are all nodes that are pointing towards x. So their weight did not contribute to Φ before and hence Φ does not decrease.

So, there is no move that decreases the value of Φ. The second statement also follows from the preceding proof, since there is no move that decreases the number of Φ-edges. □

Lemma 3. *At any time the number of Φ-edges is at most $n/2$.*

Proof. Let e_1, e_2 be two incident edges. Without loss of generality $w(e_1) > w(e_2)$. Assume that both $e_1 = \langle v_1, v_2\rangle$ and $e_2 = \langle v_2, v_3\rangle$ contribute to function Φ. If v_2 points to neither v_1 nor v_3 then both v_1 and v_3 have to point to v_2, and the move of v_2 would enable v_3, contradicting the assumption. If $v_2.p = v_1$ then e_2 can only be a Φ-edge if v_1 points to an edge heavier than e_1. So v_2 is enabled and v_1 does not point to v_2. Therefore e_1 is not a Φ-edge. If $v_2.p = v_3$ then v_2 is enabled to point to v_1 unless v_1 itself points to a heavier edge. In the latter case, there is no node pointing towards e_1 and therefore it cannot be a Φ-edge. If v_2 is enabled to point to v_1, e_2 cannot be a Φ-edge. Therefore the set of Φ-edges always forms a matching of the underlying graph. □

6.2. Complexity analysis

This section proves that Algorithm 2 stabilizes in $O(nm)$ steps under the central scheduler. For this purpose the graph G' that is obtained by removing the lightest edge from G is examined. It is shown that any execution of Algorithm 2 for G can be mapped to a closely related valid execution for G'. In particular a bound for the number of additional moves for G in comparison to G' is established. This allows us to leverage induction on the number of edges.

More precisely, it is proven that certain sequences of moves on the lightest edge increase the number of Φ-edges. Furthermore, it is shown that particular moves can only occur in the context of these sequences. Then Lemmas 2 and 3 allow us to derive an upper bound for the number of moves.

Let $\langle a, b\rangle$ be the lightest edge of G and without loss of generality $a < b$ with respect to the fixed order of the nodes. Let $G' := G\backslash\{\langle a, b\rangle\}$. We define the transformation π_c that converts a configuration of C^G into a configuration of $C^{G'}$, i.e. the states of the nodes a and b are changed to ensure that they are not pointing towards each other and they do not store the weight of edge $\langle a, b\rangle$.

$$\pi_c \; : \; \begin{cases} C^G \longrightarrow C^{G'}, \\ (s_{v_1}, \ldots, s_{v_n}) \mapsto (s'_{v_1}, \ldots, s'_{v_n}), \text{ where} \end{cases}$$

Table 1
Moves in G and corresponding moves in G'.

Move in $\mathcal{M}^G$		Move in $\mathcal{M}_\perp^{G'}$	
Name	Move	Name	Move
$\mathfrak{m}_{1a}$	$((null, *), (b, w(\langle a, b\rangle)))$	$\mathfrak{m}'_{1a}$	$\perp_0^a$
$\mathfrak{m}_{1b}$	$((null, *), (a, w(\langle a, b\rangle)))$	$\mathfrak{m}'_{1b}$	$\perp_0^b$
$\mathfrak{m}_{2a}$	$((y \neq null, *), (b, w(\langle a, b\rangle)))$	$\mathfrak{m}'_{2a}$	$((*, *), (null, 0))$
$\mathfrak{m}_{2b}$	$((y \neq null, *), (a, w(\langle a, b\rangle)))$	$\mathfrak{m}'_{2b}$	$((*, *), (null, 0))$
$\mathfrak{m}_{3a}$	$((b, *), (null, 0))$	$\mathfrak{m}'_{3a}$	$\perp_0^a$
$\mathfrak{m}_{3b}$	$((a, *), (null, 0))$	$\mathfrak{m}'_{3b}$	$\perp_0^b$
$\mathfrak{m}_{4a}$	$((b, *), (y \neq b, w(\langle a, y\rangle)))$	$\mathfrak{m}'_{4a}$	$((null, *), (y, w(\langle a, y\rangle)))$
$\mathfrak{m}_{4b}$	$((a, *), (y \neq a, w(\langle b, y\rangle)))$	$\mathfrak{m}'_{4b}$	$((null, *), (y, w(\langle b, y\rangle)))$

$$
\begin{aligned}
s'_{v_i} &= s_{v_i}, \quad \text{if } v_i \notin \{a, b\} \\
s'_a &= \begin{cases} s_a, & \text{if } s_a.p \neq b \\ (null, 0), & \text{if } s_a.p = b \wedge s_a.w = w(\langle a, b\rangle) \\ (null, s_a.w), & \text{if } s_a.p = b \wedge s_a.w \neq w(\langle a, b\rangle) \end{cases} \\
s'_b &= \begin{cases} s_b, & \text{if } s_b.p \neq a \\ (null, 0), & \text{if } s_b.p = a \wedge s_b.w = w(\langle b, a\rangle) \\ (null, s_b.w), & \text{if } s_b.p = a \wedge s_b.w \neq w(\langle b, a\rangle). \end{cases}
\end{aligned}
$$

There are four kinds of moves that may appear in an execution of Algorithm 2 for G, which cannot be executed for G'. Therefore, a mapping from the moves that are executable in G to the moves executable in G' is defined. The following shorthand notation is used throughout the rest of the paper. For $x \in \{a, b\}$ let

$$
\perp_0^x = \begin{cases} \perp, & \text{if } s_x.w = w(\langle x, s_x.p\rangle) \\ ((null, *)_x, (null, 0)_x), & \text{if } s_x.w \neq w(\langle x, s_x.p\rangle). \end{cases}
$$

Note that a node cannot be out of sync after its first move, so, with the possible exception of the case for the node's very first move, $\perp_0^x = \perp$.

Let $\mathcal{M}_\perp^{G'} := \mathcal{M}^{G'} \cup \{\perp\}$. Every move of $\mathcal{M}^G$ is mapped to a move in $\mathcal{M}_\perp^{G'}$. The only moves that will be changed are related to edge $\langle a, b\rangle$. The four kinds of moves are introduced in Table 1. Move $\mathfrak{m}_{ix}$ represents the move of type $i \in \{1, \ldots, 4\}$, performed by node $x \in \{a, b\}$. It is mapped to move $\mathfrak{m}'_{ix}$. All other moves remain unchanged, so formally we define a mapping

$$
\begin{aligned}
\pi_{\mathfrak{m}} &: \begin{cases} \mathcal{M}^G \longrightarrow \mathcal{M}_\perp^{G'}, \\ \mathfrak{m} \mapsto \mathfrak{m}', \text{ where} \end{cases} \\
\mathfrak{m}' &= \begin{cases} \mathfrak{m}'_{ix}, & \text{if } \mathfrak{m} = \mathfrak{m}_{ix}, \text{for } x \in \{a, b\} \wedge i \in \{1, \ldots, 4\} \\ \mathfrak{m}, & \text{in all other cases.} \end{cases}
\end{aligned}
$$

Moves $\mathfrak{m}_{4a}$ and $\mathfrak{m}_{4b}$ will receive a special treatment; the moves referred to as the *list moves* are those of types 1, 2 and 3 only.

Lemma 4. *If a node v in configuration c of G is enabled to perform move $\mathfrak{m}$, then v is enabled to perform move $\pi_{\mathfrak{m}}(\mathfrak{m})$ in configuration $\pi_c(c)$ of G'.*

Proof. We distinguish four possibilities for move $\mathfrak{m}$. Let $\mathfrak{m}' = \pi_{\mathfrak{m}}(\mathfrak{m})$ and $c' = \pi_c(c)$.

Case 1: $\mathfrak{m} = \mathfrak{m}_{2a}$: All neighbors of a point to other nodes via heavier edges, so a has no edge left to point to but $\langle a, b\rangle$. π_c does not change anything about that, but G' does not contain edge $\langle a, b\rangle$, so a is enabled to set its pointer to *null* instead.

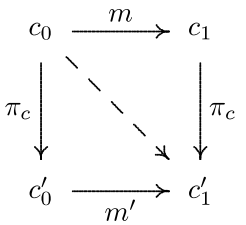
Case 2: $\mathfrak{m} \in \{\mathfrak{m}_{1a}, \mathfrak{m}_{3a}\}$: $\mathfrak{m}' = \perp_0^a$. Every node is enabled at any time with respect to the empty move, and a node that is not in sync is enabled as well.

Case 3: $\mathfrak{m} = \mathfrak{m}_{4a}$: In configuration c move $\mathfrak{m}_{4a}$ was enabled, i.e. a points to b and wants to turn its pointer towards a node x via a heavier edge. π_c sets $a.p$ to *null*. From a's point of view this does not change anything else, so move $\mathfrak{m}'_{4a}$ is enabled.

Case 4: $\mathfrak{m}$ is a move that is not contained in the list above: $\mathfrak{m}' = \mathfrak{m}$. π_c does not affect moves that are not related to edge $\langle a, b\rangle$.

The corresponding moves of node b can be treated alike. □

Lemma 5. *Let c_0 be a configuration of G and $\mathfrak{m}$ a move enabled in this configuration with respect to Algorithm 2. Then $\pi_c(m(c_0)) = m'(\pi_c(c_0))$, where $\mathfrak{m}' = \pi_{\mathfrak{m}}(\mathfrak{m})$. In other words the following diagram is commutative.*



Proof. Let x be the node that performs move $\mathfrak{m}$. We distinguish four cases:

(a) $x \notin \{a, b\}$, thus $\mathfrak{m}' = \mathfrak{m}$:

$$\begin{aligned}
\pi_c(\mathfrak{m}(c_0)) &= \pi_c(\mathfrak{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_x, \ldots, s_{v_n})))\\
&= \pi_c((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, \mathfrak{m}(s_x), \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, s'_a, \ldots, s'_b, \ldots, \mathfrak{m}(s_x), \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, s'_a, \ldots, s'_b, \ldots, \mathfrak{m}'(s_x), \ldots, s_{v_n})\\
&= \mathfrak{m}'((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_x, \ldots, s_{v_n}))\\
&= \mathfrak{m}'(\pi_c(c_0)).
\end{aligned}$$

For the rest of this proof let $x = a$.

(b) $\mathfrak{m} \in \{\mathfrak{m}_{1a}, \mathfrak{m}_{3a}\}$, so $\mathfrak{m}' = \perp_0^a$. So a either points to *null* after its move, which will remain unaffected by the application of π_c, or it points to b; in this case it will point to *null* after the application of π_c:

$$\begin{aligned}
\pi_c(\mathfrak{m}(c_0)) &= \pi_c(\mathfrak{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_{v_n})))\\
&= \pi_c((s_{v_1}, \ldots, \mathfrak{m}(s_a), \ldots, s_b, \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, \pi_c(\mathfrak{m}(s_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, \pi_c(\mathfrak{m}(s_a.p, s_a.w)), \ldots, s'_b, \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, (\mathit{null}, 0)_a, \ldots, s'_b, \ldots, s_{v_n}).
\end{aligned}$$

If node a is enabled to perform the moves $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{3a}$ in configuration c_0, it must be pointing to *null* or b before its move. Hence $\pi_c(s_a) = (\mathit{null}, 0)$ if a had its weight and its pointer in sync.

$$\begin{aligned}
\mathfrak{m}'(\pi_c(c_0)) &= \perp_0^a((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n}))\\
&= \perp_0^a((s_{v_1}, \ldots, (\mathit{null}, 0)_a, \ldots, s'_b, \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, \perp_0^a((\mathit{null}, 0)_a), \ldots, s'_b, \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, (\mathit{null}, 0)_a, \ldots, s'_b, \ldots, s_{v_n}).
\end{aligned}$$

If a did not perform a move before, then this node may be out of sync. In this case $\pi_c(s_a) = (\mathit{null}, s_a.w)$ and move $\mathfrak{m}'$ is not the empty move but it sets the weight of a to 0.

$$\begin{aligned}
\mathfrak{m}'(\pi_c(c_0)) &= \perp_0^a((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n}))\\
&= \perp_0^a((s_{v_1}, \ldots, (\mathit{null}, s_a.w)_a, \ldots, s'_b, \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, \perp_0^a((\mathit{null}, s_a.w)_a), \ldots, s'_b, \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, (\mathit{null}, 0)_a, \ldots, s'_b, \ldots, s_{v_n}).
\end{aligned}$$

(c) $\mathfrak{m} = \mathfrak{m}_{4a}$:

$$\begin{aligned}
\pi_c(\mathfrak{m}(c_0)) &= \pi_c(\mathfrak{m}((s_{v_1}, \ldots, (b, *)_a, \ldots, s_b, \ldots, s_{v_n})))\\
&= \pi_c((s_{v_1}, \ldots, \mathfrak{m}((b, *)_a), \ldots, s_b, \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, \pi_c(\mathfrak{m}((b, *)_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, \pi_c((x \neq b, w(\langle a, x\rangle))_a)), \ldots, s'_b, \ldots s_{v_n})\\
&= (s_{v_1}, \ldots, (x \neq b, w(\langle a, x\rangle))_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

and

$$\begin{aligned}
\mathfrak{m}'(\pi_c(c_0)) &= \mathfrak{m}'((s_{v_1}, \ldots, \pi_c((b, *)_a), \ldots, \pi_c(s_b), \ldots, s_{v_n}))\\
&= \mathfrak{m}'((s_{v_1}, \ldots, (\mathit{null}, 0)_a, \ldots, s'_b, \ldots, s_{v_n}))\\
&= (s_{v_1}, \ldots, \mathfrak{m}'((\mathit{null}, 0)_a), \ldots, s'_b, \ldots, s_{v_n})\\
&= (s_{v_1}, \ldots, (x \neq b, w(\langle a, x\rangle))_a, \ldots, s'_b, \ldots, s_{v_n}).
\end{aligned}$$

(d) Finally, $\mathfrak{m} = \mathfrak{m}_{2a}$.

$$\begin{aligned}\pi_c(\mathfrak{m}(c_0)) &= \pi_c(\mathfrak{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_{v_n})))\\ &= \pi_c((s_{v_1}, \ldots, \mathfrak{m}(s_a), \ldots, s_b, \ldots, s_{v_n}))\\ &= (s_{v_1}, \ldots, \pi_c(\mathfrak{m}(s_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n})\\ &= (s_{v_1}, \ldots, \pi_c(\mathfrak{m}(s_a.p, s_a.w)), \ldots, s'_b, \ldots, s_{v_n})\\ &= (s_{v_1}, \ldots, \pi_c((b, w(\langle a, b\rangle)_a), \ldots, s'_b, \ldots, s_{v_n})\\ &= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n}).\end{aligned}$$

If the configuration is transformed first it should be pointed out that a is enabled to perform move $\mathfrak{m}_{2a}$ in configuration c_0 if it points to a node other than b before. Hence, π_c does not affect s_a and $\mathfrak{m}'$ is the move that lets a point to *null*. This yields

$$\begin{aligned}\mathfrak{m}'(\pi_c(c_0)) &= \mathfrak{m}'((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n}))\\ &= \mathfrak{m}'((s_{v_1}, \ldots, s_a, \ldots, s'_b, \ldots, s_{v_n}))\\ &= (s_{v_1}, \ldots, \mathfrak{m}'(s_a), \ldots, s'_b, \ldots, s_{v_n})\\ &= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n}).\end{aligned}$$

The same arguments hold for $x = b$. Thus, in all cases $\pi_c(\mathfrak{m}(c_0)) = \mathfrak{m}'(\pi_c(c_0))$. □

Lemma 6. *If c is a legitimate configuration of Algorithm 2 for G, then $\pi_c(c)$ is a legitimate configuration for G'.*

Proof. Let c be a legitimate configuration for G. π_c cannot alter the state of any nodes but a and b. Furthermore, these nodes are only changed if one of them points to the other. Let x be a node of G. We distinguish three cases:

Case 1: $x.p = y$ and $\{x, y\} \neq \{a, b\}$.
Since c is legitimate, y points at x. None of them gets enabled by applying π_c.

Case 2: $x.p = null$.
x cannot be a neighbor of a, unless a points at a heavier edge. In none of these cases does one of the nodes involved get activated by π_c. The same holds for the neighbors of b.

Case 3: $x \in \{a, b\}$; a and b are pointing at each other.
π_c sets both nodes' pointers to *null*. Since $\langle a, b\rangle$ is not contained in G', a and b cannot point to each other. If, without loss of generality, a is enabled in configuration $\pi_c(c)$, there must be a neighbor $z \in N(a)$ in G' that points to *null* or to a. Since z remains unaffected by π_c, it must have been pointing towards a in c as well. Thus, a was enabled in c in contradiction to the assumption. □

Hence, from an execution for G we can derive an execution for G' that differs from the original execution only in the list moves. Both executions result in legitimate configurations that are related via π_c. For G the algorithm will need at most as many additional steps as there are moves replaced by move $\perp$ in G'. The latter only applies to moves of type 1 and 3.

Let $\#(\mathfrak{m}_{ix})$ denote the number of executed moves of type i by node x. Besides, let $\#(G)$ denote the number of executed moves of a given execution for graph G. This yields

$$\#(G) \leq \#(G') + \#(\mathfrak{m}_{1a}) + \#(\mathfrak{m}_{1b}) + \#(\mathfrak{m}_{3a}) + \#(\mathfrak{m}_{3b}).$$

To analyze how often the moves in question can be executed, it is shown that these moves increase the number of Φ-edges in certain situations. Lemma 3 limits the number of such edges to at most $n/2$. Therefore the executions of these moves can be bounded.

At first, it should be noted that nodes a and b cannot point to *null* at the same time, except for the initial configuration. As soon as one of them executes a move, this situation will not occur again. a and b cannot perform any of the moves of the list as long as both of them point to other nodes (the edges to these nodes are heavier). For instance, move $\mathfrak{m}_{2b}$ cannot be executed after $\mathfrak{m}_{2a}$ without having another move of b in between that makes it point to a heavier edge first.

The list moves cannot be executed in arbitrary order. For example, move $\mathfrak{m}_{3a}$ cannot be executed twice without having move $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{2a}$ in between them, since it requires a to point to b.

Definition 2. Let $\mathfrak{m}_0, \mathfrak{m}_1, \ldots, \mathfrak{m}_k$ be a sequence of moves corresponding to an execution of Algorithm 2 for graph G. If $\mathfrak{m}_i$ and $\mathfrak{m}_j$ are list moves and $\mathfrak{m}_l$ is not a list move for all l with $i < l < j$, then $[\mathfrak{m}_i, \mathfrak{m}_j]$ is called a *list free sequence*.

Table 2 shows the possible list free sequences according to Algorithm 2. Each row represents the list free sequences that start with the specified move in the first column and end with any of the moves in the following columns. We abandon the initial, nonrecurring situation in which $\mathfrak{m}_{1a}$ and $\mathfrak{m}_{1b}$ could follow after each other without one of the other list moves in between. Table 2 already contains the results of Lemma 8. The cases in which a new Φ-edge is created before the second list move becomes enabled are marked with a Φ.

In order that two moves of type 1 can be executed, there must be an intermediate move of type 2. The same holds for moves of type 3. In the following it will be shown that the number of Φ-edges increases during each of the list free sequences

Table 2
Possible list move sequences.

Name	$\mathfrak{m}_{1a}$	$\mathfrak{m}_{1b}$	$\mathfrak{m}_{2a}$	$\mathfrak{m}_{2b}$	$\mathfrak{m}_{3a}$	$\mathfrak{m}_{3b}$
$\mathfrak{m}_{1a}$	–	–	$\surd$	$\surd$	$\surd$	$\surd$
$\mathfrak{m}_{1b}$	–	–	$\surd$	$\surd$	$\surd$	$\surd$
$\mathfrak{m}_{2a}$	–	$\Phi\surd$	$\Phi\surd$	$\Phi\surd$	$\surd$	$\surd$
$\mathfrak{m}_{2b}$	$\Phi\surd$	–	$\Phi\surd$	$\Phi\surd$	$\surd$	$\surd$
$\mathfrak{m}_{3a}$	–	–	$\Phi\surd$	$\Phi\surd$	–	–
$\mathfrak{m}_{3b}$	–	–	$\Phi\surd$	$\Phi\surd$	–	–

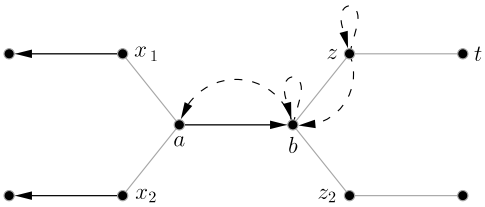


Fig. 3. Configurations of C^1.

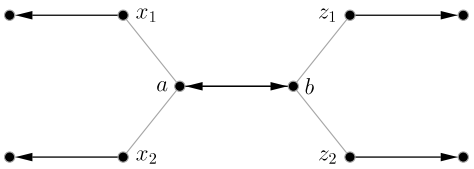


Fig. 4. Configurations of C^2.

$[\mathfrak{m}_{2a}, \mathfrak{m}_{1b}]$, $[\mathfrak{m}_{2a}, \mathfrak{m}_{2a}]$, $[\mathfrak{m}_{2a}, \mathfrak{m}_{2b}]$, $[\mathfrak{m}_{3a}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{3a}, \mathfrak{m}_{2b}]$. By symmetry the results also hold for $[\mathfrak{m}_{2b}, \mathfrak{m}_{1a}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2b}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2a}]$, $[\mathfrak{m}_{3b}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{3b}, \mathfrak{m}_{2b}]$, respectively. Two sets of configurations that will play a major role in the upcoming proofs are defined in advance:

C^1: In these configurations, node a points at b, a is disabled, i.e. all neighbors of a, except for b, point to other nodes via heavier edges. Node b points to a or to *null* and is enabled. The heaviest edge that b could point to is $\langle b, z\rangle$, where z on its part points to b or *null*. Furthermore, node z has a neighbor t. Fig. 3 shows a configuration of C^1. Note that node b (resp. z) may point to *null* or a (resp. b), which is indicated by the dotted arrows.

C^2: In these configurations, nodes a and b are pointing at each other and are disabled, i.e. all neighbors of a and b point to other nodes via heavier edges. A configuration of C^2 is shown in Fig. 4.

Remark 1. The upcoming proofs involve many nodes. Sometimes there may be several possibilities for nodes z, x, and t. For example, in Fig. 3 there are nodes x_1 and x_2. Node a might be pointing to x_1 first and then it switches to x_2 later. These moves are not of relevance – we are only interested in the list moves – so they are not considered at this point. The node that is considered to be "node x" in these cases is the one that can point to a via the heaviest edge.

Lemma 7. *If for a configuration $c \in C^1 \cup C^2$ the next list move is $\mathfrak{m}_{2a}$ (resp. $\mathfrak{m}_{2b}$), then the number of Φ-edges increases before a (resp. b) becomes enabled to perform this move.*

Proof. Let $c \in C^1$. Initially, $\langle a, b\rangle$ is not a Φ-edge. As long as this is not the case, the following holds: If an $x \in N(a)$ points towards a (necessary for $\mathfrak{m}_{2a}$ ever being executed again), then $\langle x, a\rangle$ becomes a new Φ-edge. Thus, a points at b all the time ($\mathfrak{m}_{3a}$ is not allowed), no other $x \in N(a)$ points at a and the next list move is $\mathfrak{m}_{2b}$. In order to let move $\mathfrak{m}_{2b}$ ever be executed again, b first has to point to another neighbor z. As soon as all neighbors of b (distinct from a) point to heavier edges (necessary so that $\mathfrak{m}_{2b}$ can become enabled), $\langle a, b\rangle$ becomes a new Φ-edge.

Now let $c \in C^2$. Nodes a and b are disabled and $\langle a, b\rangle$ is a Φ-edge. a (resp. b) will not become enabled unless a neighbor $x \in N(a)$ (resp. $z \in N(b)$) performs a move and points towards (a resp. b). In doing this, the Φ-edge will be moved in the corresponding direction, i.e. $\langle x, a\rangle$ (resp. $\langle z, b\rangle$) becomes a Φ-edge and $\langle a, b\rangle$ is no longer a Φ-edge, and the resulting configuration is contained in C^1 (possibly with exchanged roles of a and b). The result follows from the first case. □

Lemma 8. *During each of the list free sequences $[\mathfrak{m}_{2a}, \mathfrak{m}_{1b}]$, $[\mathfrak{m}_{2a}, \mathfrak{m}_{2a}]$, $[\mathfrak{m}_{2a}, \mathfrak{m}_{2b}]$, $[\mathfrak{m}_{3a}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{3a}, \mathfrak{m}_{2b}]$ as well as $[\mathfrak{m}_{2b}, \mathfrak{m}_{1a}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2b}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2a}]$, $[\mathfrak{m}_{3b}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{3b}, \mathfrak{m}_{2b}]$, the number of Φ-edges increases.*

Proof. The sequences will be analyzed one by one.
Case $[\mathfrak{m}_{2a}, \mathfrak{m}_{1b}]$:
Move $\mathfrak{m}_{2a}$ lets a point to b, i.e. all other neighbors of a point to heavier edges. Node b on its part cannot have been pointing to

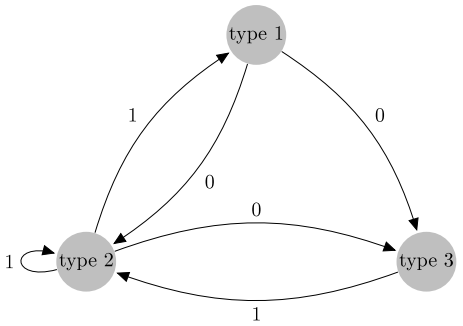


Fig. 5. Graph of list move sequences.

another node immediately before this move; otherwise $\mathfrak{m}_{2a}$ would not have been possible. Since b does not perform move $\mathfrak{m}_{3b}$ next, it must have been pointing at *null*. Assume there is no $x \in N(b)\backslash\{a\}$ that points to b or *null*, then via move $\mathfrak{m}_{2a}$ edge $\langle a, b\rangle$ would have become a new Φ-edge. So let $x \in N(b)\backslash\{a\}$ with $x.p = b$ or $x.p = null$. b cannot execute $\mathfrak{m}_{1b}$ until all of its neighbors point at a heavier edge. But in that case, again, edge $\langle a, b\rangle$ would have become a new Φ-edge. If b points to x previously, it cannot set its pointer to *null* (which is necessary for move $\mathfrak{m}_{1b}$) before a performs a move. Since other list moves are excluded, the only possibility is that a first points at a heavier edge and later to *null*. But a cannot direct its pointer to a heavier edge unless the other node of this edge that, on its part, was pointing to a heavier edge previously points to a. In doing this, a new Φ-edge is created.
Cases $[\mathfrak{m}_{2a}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{2a}, \mathfrak{m}_{2b}]$:
Node a points at b; all other neighbors of a are pointing at heavier edges. b points at a or at *null*.

- If there is a $z \in N(b)\backslash\{a\}$, which enables b, then the configuration is contained in C^1 and the rest follows from Lemma 7.
- All neighbors of b (except for a) are pointing at heavier edges. If b points at *null*, then via the first move, $\mathfrak{m}_{2a}$, $\langle a, b\rangle$ already became a new Φ-edge. If b points at a, then the configuration is contained in C^2 and the rest follows from Lemma 7.

Case $[\mathfrak{m}_{3a}, \mathfrak{m}_{2a}]$:
Initially a points at b; b on its part (like all other neighbors of a) has selected another node. Hence, a sets its pointer to *null* via executing move $\mathfrak{m}_{3a}$. In order that $\mathfrak{m}_{2a}$ can be the next list move, b has to point to *null* first (b cannot point towards a without performing a list move). This is impossible until a directs its pointer to a heavier edge first. Therefore there must be an $x \in N(a)$ that points at a before. This makes edge $\langle x, a\rangle$ a new Φ-edge.
Case $[\mathfrak{m}_{3a}, \mathfrak{m}_{2b}]$:
Again, initially a points at b; b on its part (and all other neighbors of a) has selected another node. Hence, a sets its pointer to *null* via executing move $\mathfrak{m}_{3a}$. In order that $\mathfrak{m}_{2b}$ can be the next list move, a has to point to *null* at that time. Before, the node could point to a heavier edge and back to *null*. Since this requires a node x to point towards a via a heavier edge (this node therefore would be disabled after this move) this would make $\langle x, a\rangle$ a new Φ-edge. So from now on let a point to *null* and let no other neighbor enable it. If all neighbors of b (except for a) point to a heavier edge (this is required for $\mathfrak{m}_{2b}$ being enabled), edge $\langle a, b\rangle$ becomes a new Φ-edge.

By symmetry Lemma 8 also holds for the list free sequences $[\mathfrak{m}_{2b}, \mathfrak{m}_{1a}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2b}]$, $[\mathfrak{m}_{2b}, \mathfrak{m}_{2a}]$, $[\mathfrak{m}_{3b}, \mathfrak{m}_{2a}]$ and $[\mathfrak{m}_{3b}, \mathfrak{m}_{2b}]$. □

Fig. 5 illustrates the possible list move sequences. According to Lemma 8 (cf. Table 2), edges that increase the number of Φ-edges are weighted 1, and other edges have weight 0. The basic idea of the following proof is to determine at which length a path in the depicted graph necessarily exceeds the cost of $n/2$.

Theorem 1. *Running under a central scheduler, Algorithm 2 stabilizes after at most* $(n + 3)m$ *moves if* $m > 0$.

Proof. To make use of induction it is necessary to determine an upper bound on the number of moves of types 1 and 3. Considering the graph depicted in Fig. 5, this means identifying a path not exceeding $n/2$ cost containing as many occurrences of types 1 and 3 as possible.

Since initially both nodes, a and b, could point to *null* for once, it is possible that $\mathfrak{m}_{1a}$ and $\mathfrak{m}_{1b}$ are executed without another intermediate list move. In this case the number of Φ-edges does not necessarily increase (cf. page 5535). After that the list moves follow the graph of Fig. 5. The next move of type 3 can be executed without cost increases. The following list move will be attended by an increase of Φ and so will at least every second subsequent list move.

Therefore $\#(\mathfrak{m}_{1a}) + \#(\mathfrak{m}_{1b}) + \#(\mathfrak{m}_{3a}) + \#(\mathfrak{m}_{3b}) \leq \#(\textit{all list moves}) \leq n + 3$. So, the total number of moves that the algorithm needs to stabilize in G is greater than the number of moves in G' by at most $n + 3$ moves. Clearly, for a graph that contains only one edge, Algorithm 2 stabilizes after at most n moves. By induction over the edges of G this results in $\#(G) \leq (n + 3)m$. □

In the following a lower bound for the number of moves for Algorithm 2 of $\Omega(n^2)$ for the central scheduler is provided. For this purpose consider the *line graph* with n nodes. The nodes are arranged along a line with ascending weighted edges from left to right. Initially, let all nodes point to their left neighbor, except for the first and the last node that point to *null*. Consider two phases:

(a) From left to right, all enabled nodes, one after the other, point to their right neighbor; the last one points to its left neighbor.
(b) From left to right, all enabled nodes, one after the other, point to their left neighbor; the first one points to *null*.

These two phases alternate until the algorithm stabilizes. The final configuration is reached after $n^2/2 - n/2 + 1$ moves.

7. A distributed scheduler

In [8] Gradinariu and Tixeuil showed how to transform an algorithm that stabilizes under a central scheduler into an algorithm that stabilizes under a distributed scheduler, provided unique node identifiers exist. We will now apply their method to Algorithm 2.

The basic idea is to provide the nodes with an additional variable *want_to_act*, indicating whether a node is enabled with respect to the original algorithm, and a predicate *allowed_to_act*, that is used to guarantee that no two neighboring enabled nodes execute a move simultaneously. In particular, only the node with the highest identifier among the neighboring nodes having their *want_to_act* variable set to true is allowed to execute a rule of the original algorithm.

One important property of Algorithm 2 is that a node is always disabled immediately after its move if the algorithm is running under a central scheduler. The transformation of [8] is changed in order to preserve this property: when a node executes a move of the original algorithm, simultaneously the variable *want_to_act* is set to false. This simplifies the analysis of Algorithm 3.

Algorithm 3 Self-Stabilizing 2-Approximation Maximum Weight Matching under a Distributed Scheduler

Predicates:
$allowed\_to\_act_i$:
$want\_to\_act_i \wedge i > \max\{j \in N(i) \mid want\_to\_act_j\}$
Functions:
$BestMatch(v)$:
if $C_v \neq \emptyset$ **then return** $maxC_v$
else return *null*

Actions:
$R_1 :: [want\_to\_act_i \neq (v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle))] \longrightarrow$
$want\_to\_act_i := (v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle))$
$R_2 :: [allowed\_to\_act_i] \longrightarrow$
$v.p := BestMatch(v), v.w := w(\langle v, v.p\rangle), want\_to\_act_i :=$ false

Using Algorithm 3, neighboring nodes cannot both execute a move of type R_2 in the same step. Therefore the results of Table 2 still hold true under a distributed scheduler. To prove this, arrange all moves that are executed simultaneously in an arbitrary sequential order and execute them one by one. Since none of these executing nodes are neighbors, their moves do not influence each other. Therefore, the moves of Algorithm 3 using the distributed scheduler can be regarded as being executed under the central scheduler. This allows us to carry over the definitions introduced in Section 6.2.

As for the central scheduler, the moves of Algorithm 3 for G will be mapped to moves for G'. A move in which a node x executes rule R_1 (resp. R_2) will be called $\mathfrak{u}_x$ (resp. $\mathfrak{m}_x$). Since the moves of types $\mathfrak{m}_a$ and $\mathfrak{m}_b$ cannot be executed simultaneously in G, all moves $\mathfrak{u}_x$, $\mathfrak{m}_x$ are mapped to themselves for $x \notin \{a, b\}$. The moves $\mathfrak{m}_a$ and $\mathfrak{m}_b$ will be transformed as in Section 6.2.

For $x \in \{a, b\}$ the move according to rule R_1 setting $want\_to\_act_x$ to true (resp. false) is denoted by $\mathfrak{u}_{x+}$ (resp. $\mathfrak{u}_{x-}$). The moves $\mathfrak{u}_{a+}$ and $\mathfrak{u}_{b+}$ have to be subjected to a detailed review. For example node a can be enabled to perform move $\mathfrak{m}_{1a}$ for G based on a previous execution of $\mathfrak{u}_{a+}$, which is illegal for G' (except for a not in sync), since a is not enabled to perform any move in G' in that case. Thus, $\mathfrak{u}_{a+}$ will be mapped to $\perp$ if a performs this move in G, because it wants to perform $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{3a}$ and is in sync. If the next move of a is $\mathfrak{u}_{a-}$, $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{3a}$, this move will also be mapped to $\perp$. If the next move $\mathfrak{m}_{next}$ of a is any other move, then it is mapped to a move consisting of a combination of $\mathfrak{u}_{a+}$ and $\mathfrak{m}_{next}$. Node b is treated alike.

The following analyzes how often node a (resp. node b) can be *enabled* to perform move $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{3a}$ (resp. $\mathfrak{m}_{1b}$ or $\mathfrak{m}_{3b}$). In combination with the number of *executions* of the moves $\mathfrak{m}_{1a}$, $\mathfrak{m}_{3a}$, and the corresponding moves of node b, the number of moves mapped to $\perp$ in G' can be determined.

Lemma 9. *If node a or node b is enabled to perform a type 1 move (resp. a type* 3 *move) and later it again gets enabled to perform the same move, then there will either be an intermediate execution of a list move (resp. of a list move type* 2*) or the number of Φ-edges increases.*

Proof. The two types are analyzed individually. Consider move $\mathfrak{m}_{1a}$ first. Initially, let node a be enabled to perform move $\mathfrak{m}_{1a}$, i.e. a points at *null*, b points at a or to *null*, and all other neighbors of a are pointing towards heavier edges. In order for node a to become enabled to perform move $\mathfrak{m}_{1a}$ anew without executing it (in this case the number of Φ-edges increases; see Table 2), $\mathfrak{m}_{1a}$ has to be disabled in the first place. This can be realized, as either b points at a heavier edge, or a neighbor of

a points at a via a heavier edge. In the former case, a cannot become enabled to perform move $\mathfrak{m}_{1a}$ unless b performs move $\mathfrak{m}_{1b}$ or $\mathfrak{m}_{2b}$ first.

So let b point to a or to *null* constantly. If a neighbor $x \in N(a)$ points at a, then $\langle x, a\rangle$ is a Φ-edge. If b was enabled before this move, this is a new Φ-edge. In particular this is the case if b points to *null*. So assume b disabled. If a neighbor $z \in N(b)$ should point at b then $\langle z, b\rangle$ would become a new Φ-edge. If x should point at a heavier edge later (required for a being enabled to perform move $\mathfrak{m}_{1a}$), then $\langle a, b\rangle$ will become a new Φ-edge.

Next move $\mathfrak{m}_{3a}$ is considered. Let node a be enabled to perform move $\mathfrak{m}_{3a}$; that is a points at b, and all neighbors of a, including b, are pointing at heavier edges. So that node a becomes enabled to perform move $\mathfrak{m}_{3a}$ again, without executing this move, first of all $\mathfrak{m}_{3a}$ has to be disabled. This can be realized, as either b or another neighbor of a points at a. If b directs its pointer towards a now, this is the move $\mathfrak{m}_{2b}$. So let b constantly point to heavier edges in the following. Two cases are considered:

Case 1: Edge $\langle a, b\rangle$ is a Φ-edge.
If a neighbor $x \in N(a)\backslash\{b\}$ points at a, edge $\langle x, a\rangle$ replaces $\langle a, b\rangle$ as a Φ-edge. In order to enable $\mathfrak{m}_{3a}$ anew, x has to point to a heavier edge again. In doing this, this edge replaces $\langle x, a\rangle$ as a ϕ-edge and $\langle a, b\rangle$ becomes a new Φ-edge.

Case 2: Edge $\langle a, b\rangle$ is not a Φ-edge.
As long as $\langle a, b\rangle$ is not a Φ-edge, the following holds true: If a neighbor $x \in N(a)\backslash\{b\}$ points at a, $\langle x, a\rangle$ becomes a new Φ-edge.

The same arguments hold for moves $\mathfrak{m}_{1b}$ and $\mathfrak{m}_{3b}$. □

Lemma 10. *Node a (resp. node b) will not be enabled to perform move $\mathfrak{m}_{1a}$ or $\mathfrak{m}_{3a}$ (resp. $\mathfrak{m}_{1b}$ or $\mathfrak{m}_{3b}$) more than $n/2 + 1$ times each.*

Proof. Before a becomes enabled to perform move $\mathfrak{m}_{1a}$ (resp. move $\mathfrak{m}_{3a}$) – except for the very first execution – the number of Φ-edges increases or a list move of type 1 or type 2 (resp. type 2) will be executed (Lemma 9). Lemma 8 yields that the number of Φ-edges increases before a new move of type 1 (resp. type 3) is possible in that situation. Thus, a will be enabled to perform move $\mathfrak{m}_{1a}$ (resp. $\mathfrak{m}_{3a}$) at most $n/2 + 1$ times. The same holds for moves $\mathfrak{m}_{1b}$ and $\mathfrak{m}_{3b}$. □

Theorem 2. *Running under a distributed unfair scheduler Algorithm 3 stabilizes after at most $(4n + 8)m$ moves.*

Proof. As in Section 6.2 a bound for the possible additional moves for graph G compared to graph G' is calculated. In the distributed case, therefore, the number of moves of type $\mathfrak{u}_{a+}$ and $\mathfrak{u}_{b+}$ that are mapped to $\perp$ have to be counted, each with at most one consecutive move. $\#(\mathfrak{u}_{a+}) \leq (n/2 + 1) + (n/2 + 1) = n + 2$ (Lemma 10). In the worst case, every time the consecutive move will be mapped to $\perp$ as well. That makes a total of $2n + 4$ moves. The same number has to be added for $\mathfrak{u}_{b+}$. This yields $\#(G) \leq \#(G') + 2(2n + 4)$. The theorem is now easily proved by induction on the number of edges of G. □

8. Conclusion

This paper presented a new analysis of the time complexity of a self-stabilizing algorithm that computes a 2-approximation for the maximum weight matching problem [17]. The analysis is based on a novel proof technique. It is shown that the original algorithm requires $O(nm)$ moves under the central scheduler and a modified version $O(nm)$ moves under the distributed scheduler. Previously known bounds were exponential. We believe that the new proof technique can be applied to other problems as well. The paper concludes with two conjectures.

Conjecture 1. *Algorithm 2 stabilizes after at most $O(n^2)$ moves under the central scheduler. This bound also holds for Algorithm 3 using the distributed scheduler.*

Conjecture 2. *Algorithm 2 stabilizes after at most $O(n^2)$ moves under the distributed scheduler even without the transformation of [8].*

The example presented at the end of Section 6 only requires $O(n^2)$ moves and we were unable to find an example requiring $O(nm)$ moves, where m is not in the order of n. It seems that in order to prove any of these conjectures, a different approach is required. Induction on the number of edges is no longer possible.

Acknowledgements

We wish to particularly acknowledge the thoughtful contributions of the anonymous reviewers who made many helpful suggestions regarding this paper.

References

[1] S. Cantarell, A.K. Datta, F. Petit, Self-stabilizing atomicity refinement allowing neighborhood concurrency, in: S.-T. Huang, T. Herman (Eds.), Self-Stabilizing Systems, in: LNCS, vol. 2704, Springer, 2003, pp. 102–112.
[2] S. Chattopadhyay, L. Higham, K. Seyffarth, Dynamic and self-stabilizing distributed matching, in: 21st Annual Symp. on Principles of distributed computing, ACM, New York, USA, 2002, pp. 290–297.
[3] S. Dolev, Self-stabilization, MIT Press, Cambridge, MA, USA, 2000.

[4] S. Dolev, R. Yagel, Self-stabilizing device drivers, in: A.K. Datta, M. Gradinariu (Eds.), Stabilization, Safety, and Security of Distributed Systems, 8th Int. Symposium, in: LNCS, vol. 4280, Springer, 2006, pp. 276–289.
[5] J. Edmonds, Paths, trees, and flowers, Canad. J. Math. 17 (1965) 449–467.
[6] M. Elkin, Distributed approximation: a survey, SIGACT News 35 (4) (2004) 40–57.
[7] M. Gradinariu, S. Tixeuil, Self-stabilizing vertex coloration and arbitrary graphs, in: F. Butelle (Ed.), OPODIS. Studia Informatica Universalis, Suger, Saint-Denis, rue Catulienne, France, 2000, pp. 55–70.
[8] M. Gradinariu, S. Tixeuil, Conflict managers for self-stabilization without fairness assumption, in: ICDCS'07: 27th Int. Conf. on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2007, p. 46.
[9] N. Guellati, H. Kheddouci, A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs, J. Parallel Distrib. Comput. (2009).
[10] S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Maximal matching stabilizes in time $O(m)$, Inf. Process. Lett. 80 (5) (2002) 221–223.
[11] T. Herman, Models of self-stabilization and sensor networks, in: S.R. Das, S.K. Das (Eds.), Distributed Computing—IWDC 2003, in: LNCS, vol. 2918, Springer, Berlin, 2004, 836–836.
[12] S.-C. Hsu, S.-T. Huang, A self-stabilizing algorithm for maximal matching, Inf. Process. Lett. 43 (2) (1992) 77–81.
[13] S. Kamei, H. Kakugawa, A self-stabilizing approximation algorithm for the distributed minimum k-domination, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E88-A (5) (2005) 1109–1116.
[14] S. Kamei, H. Kakugawa, A self-stabilizing distributed approximation algorithm for the minimum connected dominating set, in: IPDPS, IEEE, 2007, pp. 1–8.
[15] M.H. Karaata, Self-stabilizing strong fairness under weak fairness, IEEE Trans. Parallel Distrib. Syst. 12 (2001) 337–345.
[16] Z. Lotker, B. Patt-Shamir, A. Rosén, Distributed approximate matching, SIAM J. Comput. 39 (2) (2009) 445–460.
[17] F. Manne, M. Mjelde, A self-stabilizing weighted matching algorithm, in: Stabilization, Safety, and Security of Distributed Systems, 9th Int. Symposium, 2007, pp. 383–393.
[18] F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A self-stabilizing 2/3-approximation algorithm for the maximum matching problem, in: 10th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 94–108.
[19] F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A new self-stabilizing maximal matching algorithm, Theoret. Comput. Sci. 410 (14) (2009) 1336–1345.
[20] M. Nesterenko, A. Arora, Stabilization-preserving atomicity refinement, J. Parallel Distrib. Comput. 62 (5) (2002) 766–791.
[21] T. Nieberg, Local, distributed weighted matching on general and wireless topologies, in: 5th Int. Workshop on Foundations of Mobile Computing, ACM, New York, USA, 2008, pp. 87–92.
[22] R. Preis, Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs, in: STACS, 1999, pp. 259–269.
[23] G. Tel, Maximal matching stabilizes in quadratic time, Inf. Process. Lett. 49 (6) (1994) 271–272.
[24] V. Turau, Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler, Inf. Process. Lett. 103 (3) (2007) 88–93.
[25] V. Turau, C. Weyer, Fault tolerance in wireless sensor networks through self-stabilization, Int. J. Commun. Networks Distrib. Syst. 2 (1) (2009) 78–98.
[26] M. Wattenhofer, R. Wattenhofer, Distributed weighted matching, in: 18th An. Conf. on Distr. Comp., DISC, Amsterdam, 2004, pp. 335–348.
[27] R. Yagel, Self-stabilizing operating systems, Ph.D. Thesis, Ben-Gurion University, Negev, 2007.